Recap: Typical Compiler Structure

This Lecture (1)

In this lecture, we will walk through a rudimentary but complete compiler implemented in Haskell for the Trivial eXpression Language (TXL):

- concrete illustration of different compiler phases
- “mental scaffolding”
- brush up your Haskell knowledge

Informal TXL Syntax and Semantics

Some examples of TXL programs, concrete syntax, and their intended meaning, semantics:

- let x = 3 * 7 in let x = x * 3 in x - 21
  Semantics: ???

Some possibilities:

- Disallow re-definition of entities already in scope.
- Allow nested scopes, decide how to disambiguate; e.g., closest containing scope.
- Recursive definitions or not? I.e., is the defined entity in scope in its own definition?

We opt for nesting, closest containing scope, no recursion. Semantics: 42

TXL Really Is Trivial (1)

- TXL is indeed a trivial language.
- The meaning of a TXL “program” is just a number that could be computed statically, at compile time.
- Thus not really necessary to generate a program!
- Or we could generate a very simple program of the form
  print n;
TXL Really Is Trivial (2)

• But we are going to ignore this and generate a program that carries out the evaluation anyway.

Rough Concrete Compiler Design (1)

Each “box” of the compiler structure diagram corresponds to a function:

- scanner :: [Char] -> [Token]
- parser :: [Token] -> AST
- checker :: AST -> ChkdAST
- codegen :: ChkdAST -> Code

The compiler can thus be constructed by composing these functions in sequence. For example, scanning followed by parsing:

```haskell
scanAndParse :: [Char] -> AST
scanAndParse = parser . scanner
```

Rough Concrete Compiler Design (2)

By using reverse function composition, we can mirror the diagram very closely:

```haskell
-- Reverse function composition
(>>>) :: (a -> b) -> (b -> c) -> (a -> c)

compiler :: [Char] -> Code
compiler = scanner >>> parser >>> checker >>> codegen
```

What Happened to the Errors? (1)

Recall:

- source program
- target program
- error diagnostics

Any of the front-end phases could encounter errors. Thus, each function should return either the result of transformation or an error indication.

What Happened to the Errors? (2)

data Either a b = Left a | Right b

```haskell
scanner :: [Char] -> Either [Token] [ErrMsg]
parser :: [Token] -> Either AST [ErrMsg]
checker :: AST -> Either CheckedAST [ErrMsg]
codegen :: CheckedAST -> Code
```

In informal lexical aspects of TXL

If doing things properly, we would specify the lexical syntax of the basic language symbols, the tokens (like integers, identifiers, operators, keywords), using e.g. regular expressions.

Some notes about white space:

- The only significance of white space is to separate tokens.
- Comments starts with ! and runs to the end of the line.
- Comments are treated as white space.

TXL Compilation: Lexical Analysis

A small TXL program (the source code):

```haskell
let x = 3 * 7 in
let x = x * 3 in
x - 21
```

After lexical analysis (type: [Token]):

```
[T_Let, T_Id "x", T_Equal, T_Int 3, T_Times, T_Int 7, T_In, T_Let,
 T_Id "x", T_Equal, T_Id "x", T_Times, T_Int 3, T_In, T_Id "x", T_Minus,
 T_Int 21]
```

Context Free Grammar for TXL

```
TXLProgram  -->  Exp
Exp          -->  Exp + PrimExp
               |  Exp - PrimExp
               |  Exp * PrimExp
               |  Exp / PrimExp
PrimExp      -->  IntegerLiteral
               |  Identifier
               |  ( Exp )
               |  let Identifier = Exp in Exp
```

Exercise (home): Draw the derivation tree for

```haskell
let x = 3 * 7 in let x = x * 3 in x - 21
```

Compare with the Abstract Syntax Tree (AST) on next slide!
TXL Compilation: Syntactic Analysis

let \( x = 3 * 7 \) in let \( x = x * 3 \) in \( x - 21 \)

After parsing, AST drawn as a tree:

![AST Tree]

Notes on the TXL Parser

• Predictive, recursive-descent parser (see G52MAL).
• Parsing function for each non-terminal has type:
  \([\text{Token}] \rightarrow (\text{Exp}, [\text{Token}])\)
• As recursive descent parsers cannot handle left-recursion, grammar transformed first to eliminate left-recursion, then systematically transformed into a parser.

Compiling the TXL let-expression (1)

• The let-construct is an expression, and standard rules for nested scope are supposed to be respected.
• This is really the only complication when compiling TXL into C, since C does not have let-expressions or anything similar.
• In particular, it is not possible to directly replace let by assignment!

Compiling the TXL let-expression (2)

To see the last point, consider

\[
\text{let } x = 23 \text{ in } (\text{let } x = x * 3 \text{ in } x - 50) + x
\]

The meaning should again be 42.
However, assuming \( x \) is an int variable, the value of the following C expression is 88:

\[
(x = 23, (x = x * 3, x - 50)) + x
\]

(Final value of \( x \) is 69, 69 − 50 = 19, 19 + 69 = 88.)

Static Semantics

• TXL has only a single type, integer, so typing is trivial.
• Normal nested scope.
• Only defined variables may be used.
  For example, the following is invalid:
  \[
  \text{let } x = 3 \text{ in } y + x
  \]